Reasoning About an ACME Printer Case Investigation with Forensic Lucid

Serguei A. Mokhov, Joey Paquet, Mourad Debbabi
Faculty of Engineering and Computer Science
Concordia University, Montréal, Québec, Canada,
{mokhov,paquet,debbabi}@encs.concordia.ca

Abstract

In this work we model the ACME printer case incident and make its specification in Forensic Lucid, a Lucid- and intensional-logic-based programming language for cyberforensic analysis and event reconstruction specification. The printer case involves a dispute about two parties that was previously solved using the finite-state automata (FSA) approach, and now re-done in a more usable way in Forensic Lucid.

Keywords: intensional logic, intensional programming, cyberforensics, Forensic Lucid, Lucid, finite-state automata

1 Introduction

Problem Statement. The first formal approach for cyberforensic analysis and event reconstruction appeared in two papers [1, 2] by Gladyshev et al. that relies on the finite-state automata (FSA) and their transformation and operation to model evidence, witnesses, stories told by witnesses, and their possible evaluation. One of the examples the papers present is the use-case for the proposed technique – the ACME Printer Case Investigation. See [1] for the formalization using FSA by Gladyshev and the corresponding LIPS implementation. We aim at the same case to model and implement it using the new approach, which paves a way to be more friendly and usable in the actual investigator's work and serve as a basis to further development in the area.

Proposed Solution. We show the intensional approach to the problem is an asset in the field of cyberforensics as it is promising to be more practical and usable than the FSA. Since Lucid was originally designed and used to prove correctness of programming languages [3, 4, 5, 6], and is based on the temporal logic, functional and data-flow languages its implementation to backtracking in proving or disproving the evidential statements and claims in the investigation process as a evaluation of an expression that either evaluates to *true* or *false* given all the facts in the formally specified context.

From the logic perspective, it was shown one can model computations (the basic unit in the finite state machines in [1, 2]) as logic [7]. When armed with contexts as first-class values and a demand-driven model adopted in the implementation of the Lucid-family of languages [8, 9, 10, 11, 12, 13] that limits the scope of evaluation in a given set of dimensions, we come to the intensional logic and the corresponding programming artifact. In the essence, we model our forensic computation unit in the intensional logic and look for the ways to implement it in practice within an intensional programming platform [14, 8, 15]. We see a lot of potential for this work to be successful and beneficial for cyberforensics as well as intensional programming communities.

Based on the parameters and terms defined in the papers [1, 2], we have various pieces of evidence and witnesses telling their own stories of the incident. The goal is to put them together to make the description of the incident as precise as possible. To show that a certain claim may be true, the investigator has to show that there are some explanations of evidence that agrees with the claim. To disprove the

claim, the investigator has to show there is no explanation of evidence that agree with the claim [1]. The authors of the FSA approach did a proof-of-concept implementation of the algorithms in CMU Common LISP [1] that we target to improve the usability of it by re-writing it in a Lucid dialect, that we call Forensic Lucid with a possibility to construct a data-flow graph-based [16] IDE for the investigator to use and train novice investigators as an expert system. In this work we focus on the specification of hierarchical context expressions and the operators on them when modeling the examples. LIPS, unlike Lucid, entirely lacks contexts build into its logic, syntax, and semantics, thereby making the implementation of the cases more clumsy and inefficient (i.e. highly sequential). Our system [14, 8, 10, 15] (not discussed here) offers distributed demand-driven evaluation of Lucid programs in a more efficient way and is more general than LISP's compiler and run-time environment.

Intensional Logic and Programming. Intensional programming (IP) is based on intensional (or multidimensional) logics, which, in turn, are based on Natural Language Understanding (aspects, such as, time, belief, situation, and direction are considered). IP brings in *dimensions* and *context* to programs (e.g. space and time in physics or chemistry). Intensional logic adds dimensions to logical expressions; thus, a non-intensional logic can be seen as a constant or a snapshot in all possible dimensions. *Intensions are dimensions* at which a certain statement is true or false (or has some other than a Boolean value). *Intensional operators* are operators that allow us to navigate within these dimensions [17].

An Example of Using Temporal Intensional Logic. Temporal intensional logic is an extension of temporal logic that allows to specify the time in the future or in the past [17].

(1) E_1 := it is raining **here today**

Context: {`place`:**here**, `time`:**today**}

(2) E_2 := it was raining **here** *before*(**today**) = *yesterday*

(3) E_3 := it is going to rain *at* (altitude **here** + 500 m) *after*(**today**) = *tomorrow*

Let's take E_1 from (1) above. The context is a collection of the dimensions `place` and `time` with the corresponding tag values of **here** and **today**. Then let us fix **here** to **Honolulu** and assume it is a *constant*. In the month of March, 2009, with granularity of day, for every day, we can evaluate E_1 to either *true* or *false*, as shown in Figure 1. If one starts varying the **here** dimension (which could even be broken

```
Tags days in April):   1 2 3 4 5 6 7 8 9 ...
Values (raining?):     F F T T T F F F T ...
```

Figure 1: 1D example of tag-value contextual pairs

down to X, Y, Z), one gets a two-dimensional (or 4D) evaluation of E_1, as shown in Figure 2. Even

```
Place/Time  1 2 3 4 5 6 7 8 9 ...
Honolulu    F F T T T F F F T ...
New York    F F F F T T T F F ...
Ottawa      F T T T T T F F F ...
```

Figure 2: 2D example of tag-value contextual pairs

with these toy examples we can immediately illustrate the hierarchical notion of the dimensions in the context: so far the place and time we treated as atomic values fixed at days and cities. In some cases, we need finer subdivisions of the context evaluation, where time can become fixed at hour, minute, second and finer values, and so is the place broken down into boroughs, regions, streets, etc. and finally the

X, Y, Z coordinates in the Euclidean space with the values of millimeters or finer. This notion becomes more apparent and important in Forensic Lucid.

Lucid Overview. Lucid [3, 4, 5, 6, 18] is a dataflow intensional and functional programming language. In fact, it is a family of languages that are built upon intensional logic (which in turn can be understood as a multidimensional generalization of temporal logic) involving context and demand-driven parallel computation model. A program written in some Lucid dialect is an expression that may have subexpressions that need to be evaluated at certain *context*. Given the set of dimensions $D = \{dim_i\}$ in which an expression varies, and a corresponding set of indexes or *tags* defined as placeholders over each dimension, the context is represented as a set of $<dim_i : tag_i>$ mappings and each variable in Lucid, called often a *stream*, is evaluated in that defined context that may also evolve using context operators [9, 11, 19, 20]. The generic version of Lucid, the General Intensional Programming Language (GIPL) [17], defines two basic operators @ and # to navigate (switch and query) in the contexts $\mathscr{P}$. The GIPL is the first[1] generic programming language of all intensional languages, defined by the means of only two intensional operators `@` and `#`. It has been proven that other intensional programming languages of the Lucid family can be translated into the GIPL [17].

2 Forensic Lucid

This section summarizes concepts and considerations in the design of the Forensic Lucid language, which is being studied through another use-case than related works [21, 22, 23]. The end goal is to define our Forensic Lucid language where its constructs concisely express cyberforensic evidence as context of evaluations, which can be initial state of the case towards what we have actually observed as a final state in the FSM. The implementing system [8, 14, 15] backtraces intermediate results to provide the corresponding event reconstruction path if it exists (which we do not discuss in this work). The result of the expression in its basic form is either *true* or *false*, i.e. "guilty" or "not guilty" given the evidential evaluation context per explanation with the backtrace. There can be multiple backtraces, that correspond to the explanation of the evidence (or lack thereof).

Properties. We define Forensic Lucid to model the evidential statements and other expressions representing the evidence and observations as context. An execution trace of a running Forensic Lucid program is designed to expose the possibility of the proposed claim with the events that lead to the conclusion. Forensic Lucid aggregates the features of multiple Lucid dialects mentioned earlier needed for these tasks along with its own extensions. The addition of the context calculus from Lucx (stands for "Lucid enriched with context" that promotes contexts as first-class values) for operators on simple contexts and context sets (`union`, `intersection`, etc.) are used to manipulate complex hierarchical context spaces in Forensic Lucid. Additionally, Forensic Lucid inherits many of the properties of Lucx, Objective Lucid, JOOIP (Java-embedded Object-Oriented Intensional Programming language), and their comprising dialects, where the former is for the context calculus, and the latter for the arrays and structural representation of data for modeling the case data structures such as events, observations, and groupings of the related data, and so on. (We eliminate the OO-related aspects from this work as well as some others to conserve space and instead focus on the context hierarchies, syntax, and semantics.) Hierarchical contexts are also following the example of MARFL [24] using a dot operator and by overloading @ and # to accept different types as their left and right arguments. One of the basic requirements is that the final target definition of the syntax, and the operational semantics of Forensic Lucid should be compatible

[1] The second being Lucx [11, 9, 20]

with the basic Lucx and GIPL. This is necessary for compiler and and the run time system within the implementing system, called General Intensional Programming System (GIPSY) [8, 14]. The translation rules or equivalent are to be provided when implementing the language compiler within GIPSY, and such that the run-time environment (General Eduction Engine, or GEE) can execute it with minimal changes to GEE's implementation.

Context. We need to provide an ability to encode the stories told by the evidence and witnesses. This will constitute the context of evaluation. The return value of the evaluation would be a collection of back-traces, which contain the "paths of truth". If a given trace contains all truths values, it's an explanation of a story. If there is no such a path, i.e. the trace, there is no enough supporting evidence of the entire claim to be true. The context for this task for simplicity of the prototype language can be expressed as integers or strings, to which we attribute some meaning or description. The contexts are finite and can be navigated through in both directions of the index, potentially allowing negative tags in our tag sets of dimensions. Concurrently, our contexts can be a finite set of symbolic labels and their values that can internally be enumerated. The symbolic approach is naturally more appropriate for humans and we have a machinery to so in Lucx's implementation in GIPSY [19, 19]. We define streams of observations as our context, that can be a simple context or a context set. In fact, in Forensic Lucid we are defining higher-level dimensions and lower-level dimensions. The highest-level one is the *evidential statement*, which is a finite unordered set of observation sequences. The *observation sequence* is a finite *ordered* set of observations. The *observation* is an "eyewitness" of a particular property along with the duration of the observation. As in the FSA [2, 1], the observations are tuples of (P, min, opt) in their generic form. The observations in this form, specifically, the property P can be exploded further into Lucx's context set and further into an atomic simple context [20, 9, 19]. Context switching between different observations is done naturally with the Lucid @ context switching operator. Consider some conceptual expression of a storyboard in Listing 1 where anything in `[ ... ]` represents a story, i.e. the context of evaluation. `foo` can be evaluated at multiple contexts (stories), producing a collection of final results (e.g. *true* or *false*) for each story as well as a collection of traces.

```
foo @
{
  [ final observed event, possible initial observed event ],
  [                      ],
  [                      ]
}
```

Listing 1: Intensional Storyboard Expression

While the `[...]` notation here may be confusing with respect to the notation of `[dimension:tag]` in Lucid and more specifically in Lucx [20, 19, 9], it is in fact a simple syntactical extension to allow higher-level groups of contexts where this syntactical sugar is later translated to the baseline context constructs. The tentative notation of `{[...],...,[...]}` implies a notion similar to the notion of the "context set" in [20, 19] except with the syntactical sugar mentioned earlier where we allow syntactical grouping of properties, observations, observation sequences, and evidential statements as our context sets. The generic observation sequence [1] can be expanded into the context stream using the *min* and *opt* values, where they will translate into index values. Thus, $obs = (A,3,0)(B,2,0)$ expands the property labels A and B into a finite stream of five indexed elements: *AAABB*. Thus, a Forensic Lucid fragment in Listing 2 would return the third A of the *AAABB* context stream in the observation portion of o. Therefore, possible evaluations to check for the properties can be as shown in Figure 3. The property values of A and B can be anything that context calculus allows. The `observation sequence` is a finite ordered

```
// Give me observed property at index 2 in the observation sequence obs
o @.obs 2
where
  // Higher-level dimension in the form of (P,min,opt)
  observation o;
  // Equivalent to writing = { A, A, A, B, B };
  observation sequence obs = (A,3,0)(B,2,0);
  where
    // Properties A and B are arrays of computations
    // or any Expressions
    A = [c1,c2,c3,c4];
    B = E;
    ...
  end;
end;
```

Listing 2: Observation Sequence With Duration

context tag set [19] that allows an integral "duration" of a given tag property. This may seem like we allow duplicate tag values that are unsound in the classical Lucid semantics; however, we find our way around little further in the text with the implicit tag index. The semantics of the arrays of computations is not a part of either GIPL or Lucx; however, the arrays are provided by JLucid and Objective Lucid. We need the notion of the arrays to evaluate multiple computations at the same context. Having an array of computations is conceptually equivalent of running an a Lucid program under the same context for each array element in a separate instance of the evaluation engine and then the results of those expressions are gathered in one ordered storage within the originating program. Arrays in Forensic Lucid are needed to represent a set of results, or *explanations* of evidential statements, as well as denote some properties of observations. We will explore the notion of arrays in Forensic Lucid much greater detail in the near future work. In the FSA approach computations c_i correspond to the state q and event i that enable transition. For Forensic Lucid, we can have c_i as theoretically any Lucid expression E.

```
Observed property (context): A A A B B
        Sub-dimension index: 0 1 2 3 4

o @.obs 0 = A
o @.obs 1 = A
o @.obs 2 = A
o @.obs 3 = B
o @.obs 4 = B

To get the duration/index position:

o @.obs A = 0 1 2
o @.obs B = 3 4
```

Figure 3: Handling Duration of an Observed Property in the Context

In Figure 3 we are illustrating a possibility to query for the sub-dimension indices by raw property where it persists that produces a finite stream valid indices that can be used in subsequent expressions, or, alternatively by supplying the index we can get the corresponding raw property at that index. The latter feature is still under investigation of whether it is safe to expose it to Forensic Lucid programmers

or make it implicit at all times at the implementation level. This is needed to remedy the problem of "duplicate tags": as previously mentioned, observations form the context and allow durations. This means multiple duplicate dimension tags with implied subdimension indexes should be allowed as the semantics of a traditional Lucid approaches do not allow duplicate dimension tags. It should be noted however, that the combination of the tag and its index in the stream is still unique and can be folded into the traditional Lucid semantics.

Transition Function. A transition function (described at length [1, 2] and the derived works) determines how the context of evaluation changes during computation. A general issue exists that we have to address is that the transition function ψ is usually problem-specific. In the FSA approach, the transition function is the labeled graph itself. In the first prototype, we follow the graph to model our Forensic Lucid equivalent. In general, Lucid has already basic operators to navigate and switch from one context to another, which represent the basic transition functions in themselves (the intensional operators such as @, #, `iseod`, `first`, `next`, `fby`, `wvr`, `upon`, and `asa` as well as their inverse operators[2]). However, a specific problem being modeled requires more specific transition function than just plain intensional operators. In this case the transition function is a Forensic Lucid function where the matching state transition modeled through a sequence of intensional operators. In fact, the forensic operators are just pre-defined functions that rely on the traditional and inverse Lucid operators as well as context switching operators that achieve something similar to the transitions. At the implementation level, it is the GEE that actually does the execution of ψ within GIPSY. In fact, the intensional operators of Lucid represent the basic building blocks for ψ and Ψ^{-1}.

Generic Observation Sequences. We adopt a way of modeling generic observation sequences as a box operator from the Lucx's context calculus [20, 19] in the dimensional context that defines the space of all possible evaluations. The generic observation sequence context contains observations whose properties' duration is not fixed to the *min* value as in $(P, min, 0)$ as we studied so far. The third position in the observation tuple, opt is not 0 in the generic observation and as a result in the containing observation sequence, e.g. $os = (P_1,1,2)(P_2,1,1)$. Please refer to [1, 2, 22, 21] for more detailed examples of a generic observation sequence.

Primitive Operators. The basic set of the classic intensional operators is extended with the similar operators, but inverted in one of their aspects: either negation of trueness or reverse of direction of navigation. Here we provide a definition of these operators alongside with the classical ones (to remind the reader what they do and enlighten the unaware reader). The reverse operators have a restriction that they must work on the bounded streams at the positive infinity. This is not a stringent limitation as the our contexts of observations and evidence in this work are always finite, so they all have the beginning and the end. What we need is an ability to go back in the stream and, perhaps, negate in it with classical-like operators, but reversed. Following the steps in [17], we further represent the definition of the operators via @ and #. Again, there is a mix of classical operators that were previously defined in [17], such as `first`, `next`, `fby`, `wvr`, `upon`, and `asa` as well as the new operators from this work.

Forensic Operators. The operators presented here are based on the discussion of the combination function and others that form more-than-primitive operations to support the required implementation. The discussed earlier `comb()` operator needs to be realized in the general manner for combining analogies of MPRs, which in our case are higher-level contexts, in the new language's dimension types.

[2] Defined further

```
acmepsi(c, s, d) =
  // Add a print job from Alice
  if c == ``add_A'' then
    if d1 == ``A'' || d2 == ``A'' then s;
    else
      if d1 in S then ``A'' fby.d d2;
      else
        if d2 in S then d1 fby.d ``A'';
        else s;
  // Add a print job from Bob
  else if c == ``add_B'' then
    if d1 == ``B'' || d2 == ``B'' then s;
    else
      if d1 in S then ``B'' fby.d d2;
      else
        if d2 in S then d1 fby.d ``B'';
        else s;
  // Printer takes the job per manufacturer specification
  else if c == ``take''
    if d1 == ``A'' then ``A_deleted'' fby.d d2;
    else
      if d1 == ``B'' then ``B'' fby.d d2;
      else
        if d2 == ``A'' then d1 fby.d ``A_deleted'';
        else
          if d2 == ``B'' then d1 fby.d ``B_deleted'';
          else s;
  // Done
  else s fby.d eod;

  where
    dimension d;
    S = [``empty'', ``A_deleted'', ``B_deleted''];
    d1 = first.d s;
    d2 = next.d d1;
  end;
```

Listing 3: "Transition Function" ψ Implemented in Forensic Lucid for the ACME Printing Case

- `combine` corresponds to the *comb* function described earlier. It is defined in Listing 6. It is a preliminary context-enhanced version.
- `product` tentatively corresponds to the cross-product of context, translated from that of the LISP example and added with context. It is defined in Listing 7.

The translated examples show recursion that we are not prepared to deal with in the current Lucid semantics, and will address that in the future work. The two illustrated operators are the first of the a few more to follow in the final language prototype.

3 Conclusion

The proposed practical approach in the cyberforensics field can also be used in a normal investigation process involving crimes not necessarily associated with information technology. Combined with an expert system (e.g. implemented in CLIPS [25]), it can also be used in training new staff in investigation techniques. The focus on hierarchical contexts as first-class values brings more understanding of the process to the investigators in cybercrime case management tools.

Future Work.

```
invpsiacme(s, d) = backtraces
where
  backtraces = [A, B, C, D, E, F, G, H, I, J, K, L, M ];
  where
    A = if d1 == ``A_deleted'' then d2 pby.d ``A'' pby.d ``take''
        else eod;

    B = if d1 == ``B_deleted'' then d2 pby.d ``B'' pby.d ``take''
        else eod;

    C = if d2 == ``A_deleted'' && d1 != ``A'' && d2 != ``B''
        then d1 pby.d ``A'' pby.d ``take''
        else eod;

    D = if d2 == ``B_deleted'' && d1 != ``A'' && d2 != ``B''
        then d1 pby.d ``B'' pby.d ``take''
        else eod;

    E = if d1 in S && d2 in S then s pby.d ``take''
        else eod;

    F = if d1 == ``A'' && d2 != ``A''
        then
          [ d2 pby.d ``empty'' pby.d ``add_A'',
            d2 pby.d ``A_deleted'' pby.d ``add_A'',
            d2 pby.d ``B_deleted'' pby.d ``add_A'' ]
        else eod;

    G = if d1 == ``B'' && d2 != ``B''
        then
          [ d2 pby.d ``empty'' pby.d ``add_B'',
            d2 pby.d ``A_deleted'' pby.d ``add_B'',
            d2 pby.d ``B_deleted'' pby.d ``add_B'' ]
        else eod;

    H = if d1 == ``B'' && d2 == ``A''
        then
          [ d1 pby.d ``empty'' pby.d ``add_A'',
            d1 pby.d ``A_deleted'' pby.d ``add_A'',
            d1 pby.d ``B_deleted'' pby.d ``add_A'' ]
        else eod;

    I = if d1 == ``A'' && d2 == ``B''
        then
          [ d1 pby.d ``empty'' pby.d ``add_B'',
            d1 pby.d ``A_deleted'' pby.d ``add_B'',
            d1 pby.d ``B_deleted'' pby.d ``add_B'' ]
        else eod;

    J = if d1 == ``A'' || d2 == ``A'' then s pby.d ``add_A''
        else eod;

    K = if d1 == ``A'' && d2 == ``A'' then s pby.d ``add_B''
        else eod;

    L = if d1 == ``B'' && d2 == ``A'' then s pby.d ``add_A''
        else eod;

    M = if d1 == ``B'' || d2 == ``B'' then s pby.d ``add_B''
        else eod;

    where
      dimension d;
      S = [``empty'', ``A_deleted'', ``B_deleted''];
      d1 = first.d s;
      d2 = next.d d1;
    end;
```

Listing 4: "Inverse Transition Function" Ψ^{-1} Implemented in Forensic Lucid for the ACME Printing Case

```
alice_claim @ es
where
  evidential statement es = [ printer, manuf, alice ];

  observation sequence printer = F;
  observation sequence manuf = [Oempty, $];
  observation sequence alice = [Oalice, F];

  observation F = (``B_deleted'', 1, 0);
  observation Oalice = (P_alice, 0, +inf);
  observation Oempty = (``empty'', 1, 0);

  // No ``add_A''
  P_alice = unordered {``add_B'', ``take''};

  invpsiacme(F, es);
end;
```

Listing 5: Developing the Pinter Case Example 3

```
/**
 * Append given e to each element
 * of a given stream e under the
 * context of d.
 *
 * @return the resulting combined stream
 */
combine(s, e, d) =
  if iseod s then eod;
  else (first s fby.d e) fby.d combine(next s, e, d);
```

Listing 6: The `combine` Operator

- Forensic Lucid Compiler and run-time environment.
- Prove equivalence to the FSA approach.
- Adapt/re-implement a graphical UI based on the data-flow graph tool [26] to simplify Forensic Lucid programming for not very tech-savvy investigators.
- Refine the semantics of Lucx's context sets and their operators to be more sound, including Box and Range.

Acknowledgments. This research work was funded by the Faculty of Engineering and Computer Science of Concordia University, Montreal, Canada.

```
/**
 * Append elements of s2 to element of s1
 * in all possible combinations.
 */
product(s1, s2, d) =
  if iseod s2 then eod;
  else combine(s1, first s2) fby.d product(s1, next s2)
```

Listing 7: The `product` Operator

References

[1] Pavel Gladyshev and Ahmed Patel. Finite state machine approach to digital event reconstruction. *Digital Investigation Journal*, 2(1), 2004.

[2] Pavel Gladyshev. Finite state machine analysis of a blackmail investigation. *International Journal of Digital Evidence*, 4(1), 2005.

[3] Edward A. Ashcroft and William W. Wadge. Lucid - a formal system for writing and proving programs. *SIAM J. Comput.*, 5(3), 1976.

[4] Edward A. Ashcroft and William W. Wadge. Erratum: Lucid - a formal system for writing and proving programs. *SIAM J. Comput.*, 6((1):200), 1977.

[5] William Wadge and Edward Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, London, 1985.

[6] Edward Ashcroft, Anthony Faustini, Raganswamy Jagannathan, and William Wadge. *Multidimensional, Declarative Programming*. Oxford University Press, London, 1995.

[7] René Lalement. *Computation as Logic*. Prentice Hall, 1993. C.A.R. Hoare Series Editor. English translation from French by John Plaice.

[8] Joey Paquet and Ai Hua Wu. GIPSY – a platform for the investigation on intensional programming languages. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*, pages 8–14, Las Vegas, USA, June 2005. CSREA Press.

[9] Joey Paquet, Serguei A. Mokhov, and Xin Tong. Design and implementation of context calculus in the GIPSY environment. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, pages 1278–1283, Turku, Finland, July 2008. IEEE Computer Society.

[10] Joey Paquet. A multi-tier architecture for the distributed eductive execution of hybrid intensional programs. In *Proceedings of 2nd IEEE Workshop in Software Engineering of Context Aware Systems (SECASA'09)*. IEEE Computer Society, 2009. To appear.

[11] Kaiyu Wan, Vasu Alagar, and Joey Paquet. Lucx: Lucid enriched with context. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*, pages 48–14, Las Vegas, USA, June 2005. CSREA Press.

[12] John Plaice, Blanca Mancilla, Gabriel Ditu, and William W. Wadge. Sequential demand-driven evaluation of eager TransLucid. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, pages 1266–1271, Turku, Finland, July 2008. IEEE Computer Society.

[13] Toby Rahilly and John Plaice. A multithreaded implementation for TransLucid. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, pages 1272–1277, Turku, Finland, July 2008. IEEE Computer Society.

[14] The GIPSY Research and Development Group. The General Intensional Programming System (GIPSY) project. Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2002–2008. `http://newton.cs.concordia.ca/~gipsy/`, last viewed April 2008.

[15] Serguei A. Mokhov. Towards hybrid intensional programming with JLucid, Objective Lucid, and General Imperative Compiler Framework in the GIPSY. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, October 2005. ISBN 0494102934.

[16] Yi Min Ding. Bi-directional translation between data-flow graphs and Lucid programs in the GIPSY environment. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2004.

[17] Joey Paquet. *Scientific Intensional Programming*. PhD thesis, Department of Computer Science, Laval University, Sainte-Foy, Canada, 1999.

[18] Edward A. Ashcroft and William W. Wadge. Lucid, a nonprocedural language with iteration. *Communication of the ACM*, 20(7):519–526, July 1977.

[19] Xin Tong. Design and implementation of context calculus in the GIPSY. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, April 2008.

[20] Kaiyu Wan. *Lucx: Lucid Enriched with Context*. PhD thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2006.

[21] Serguei A. Mokhov and Joey Paquet. Formally specifying and proving operational aspects of Forensic Lucid in Isabelle. Technical Report 2008-1-Ait Mohamed, Department of Electrical and Computer Engineering, Concordia University, August 2008. In Theorem Proving in Higher Order Logics (TPHOLs2008): Emerging Trends Proceedings.

[22] Serguei A. Mokhov, Joey Paquet, and Mourad Debbabi. Formally specifying operational semantics and language constructs of Forensic Lucid. In Oliver Göbel, Sandra Frings, Detlef Günther, Jens Nedon, and Dirk Schadt, editors, *Proceedings of the IT Incident Management and IT Forensics (IMF'08)*, pages 197–216, Mannheim, Germany, September 2008. GI. LNI140.

[23] Serguei A. Mokhov. Encoding forensic multimedia evidence from MARF applications as Forensic Lucid expressions. In *Proceedings of CISSE'08*, University of Bridgeport, CT, USA, December 2008. Springer. To appear.

[24] Serguei A. Mokhov. Towards syntax and semantics of hierarchical contexts in multimedia processing applications using MARFL. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, pages 1288–1294, Turku, Finland, July 2008. IEEE Computer Society.

[25] Gary Riley. CLIPS: A tool for building expert systems. [online], 2007. `http://www.ghg.net/clips/CLIPS.html`, last viewed: December 2007.

[26] Lei Tao. Warehouse and garbage collection in the GIPSY environment. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2004.